

Monotonic Abstraction Techniques: from Parametric to Software Model Checking

Francesco Alberti

University of Lugano,
Lugano, Switzerland

Silvio Ghilardi

Università degli Studi di Milano,
Milano, Italy

Natasha Sharygina

University of Lugano,
Lugano, Switzerland

Monotonic abstraction is a technique introduced in model checking parameterized distributed systems in order to cope with transitions containing global conditions within guards. The technique has been re-interpreted in a declarative setting in previous papers of ours and applied to the verification of fault tolerant systems under the so-called ‘stopping failures’ model. The declarative reinterpretation consists in logical techniques (quantifier relativizations and, especially, quantifier instantiations) making sense in a broader context. In fact, we recently showed that such techniques can over-approximate array accelerations, so that they can be employed as a meaningful (and practically effective) component of CEGAR loops in software model checking too.

1 Introduction

Monotonic abstraction is a well-known technique introduced by P. A. Abdulla and collaborators in a series of papers (like for instance [1, 3, 8, 9]); the technique was originally applied in the context of verification of distributed systems, but successively extended also elsewhere (see e.g. [6, 7]). The approach has been reformulated in [16, 17] within the declarative context of array-based systems [34] in order to apply it to the verification of reliable broadcast algorithms [39] in a fault-tolerant environment. The declarative reformulation makes clear that monotonic abstraction can be viewed operationally as a purely symbolic manipulation applying quantifier instantiation in order to overapproximate sets of states represented via $\exists^*\forall$ -formulae. Since the same kind of formulae arise when computing loop accelerations in sequential programs for arrays [18], it is natural to import it in software model checking and to merge it with interpolation-based abstraction-refinement techniques [13, 15, 38]. This strategy has been successfully implemented in the latest versions of the model checker MCMT (see <http://users.mat.unimi.it/users/ghilardi/mcmt>). In this contribution, we describe the above reinterpretation of monotonic abstraction; we won’t enter into technicalities and we shall keep the exposition at intuitive and informal level (readers interested in details can consult the published papers mentioned below).

2 Monotonic Abstraction

We quote the content of this Section (a brief description monotonic abstraction) directly from the tutorial [10].

“A parameterized system consists of an arbitrary number number of processes usually organized as a linear array. In fact, a parameterized system represents an infinite family of systems, namely one for each size of the system. We are interested in *parameterized verification*, i.e., verifying correctness regardless of the number of processes inside the system. The term *parameterized* refers to the fact that the size of the system is (implicitly) a parameter of the verification problem. Examples of parameterized systems

include mutual exclusion algorithms, bus protocols, telecommunication protocols, and cache coherence protocols. (...)

There has been an extensive research on the verification of infinite-state systems which are *monotonic* w.r.t. a *well quasi-ordering* on the set of configurations [2]. The main idea is to perform symbolic backward reachability analysis to check safety properties for such systems. The method was first reported in [4] and applied to analyze safety properties for lossy channel systems. Concretely, we define a pre-order \preceq on the set of configurations such that (1) \preceq is a simulation with respect to the transition relation (i.e., the transition relation is monotonic w.r.t. \preceq), and (2) \preceq is a well-quasi ordering (WQO for short). Given such a pre-order, we can derive a backward algorithm for checking reachability of sets of configurations which are upward closed w.r.t. \preceq . Upward closed sets are attractive to use in this setting for several reasons. First, we are interested in safety properties, in which we check the reachability of a set of *bad configurations*. These are configurations which we do not want to occur during the execution of the system. For instance, in mutual exclusion protocols, the bad configurations are those in which at least two processes are in their critical sections. This means that checking safety properties amounts to checking reachability of upward closed sets of configurations. The second attractive feature of upward closed sets is that they can be characterized by their minimal elements, which often makes it possible to have efficient symbolic representations of infinite sets of configurations.

We start from the set of bad configurations, and then compute the sets of predecessors, i.e., sets of configurations which correspond to going one step backwards along the transition relation. Monotonicity implies that, for any upward-closed set, the set of its predecessors is an upward-closed set. Since the set of bad configurations is upward closed, it follows that all the sets which are generated are also upward closed. This procedure is guaranteed to terminate by the well quasi-ordering of the relation on the set of configurations.

Since its first application to lossy channel systems [4], the method has been used for the design of verification algorithms for a wide range of models such as Petri nets, timed Petri nets, broadcast protocols, cache coherence protocols, etc. (see, e.g., [5, 30–32]).

Unfortunately, parameterized systems do not quite fit into this framework, in the sense that there is no nontrivial (useful) WQO for which these systems are monotonic. The ordering \preceq amounts to the subword relation on words. The main obstacle is that parameterized systems usually use universal global conditions in which a process may need to check states of all other processes inside the system. Universal conditions are inherently non-monotonic, since having larger configurations may lead to the violation of the universal condition. In this tutorial, we give an overview of the method of *monotonic abstraction* [3, 7–9], which attempts to overcome this problem by defining an abstract semantics which forces monotonicity. Basically, the idea is to consider that a transition is possible from a configuration c_1 to c_2 if it is possible from any smaller configuration $c'_1 \preceq c_1$ to c_2 . More precisely, the abstraction kills (deletes) all the processes inside the configuration which violate the universal condition. Since the abstract transition relation is an over-approximation of the original one, proving a safety property in the abstract system implies that the property also holds in the original system.”

3 Array-Based Systems

Array-based systems were first introduced in [34]: the underlying idea is that of specifying systems of various nature using array theories like those studied in [24, 25] and implemented in common SMT-solvers. Array theories are multi-sorted and very flexible; typically one has three sorts: for indexes, elements and arrays. To specify a distributed system, one can use the index sort to model processes and

```

function BReach( $\mathcal{S}$ )
   $i \leftarrow 0$ ;  $BR^0(\tau, U) \leftarrow U$ ;  $K^0 \leftarrow U$ 
  if check( $BR^0(\tau, U) \wedge I$ ) = sat then return unsafe
  repeat
     $K^{i+1} \leftarrow \text{Pre}(\tau, K^i)$ 
    if check( $K^{i+1} \wedge I$ ) = sat then return unsafe
    else  $i \leftarrow i + 1$ 
     $BR^{i+1}(\tau, U) \leftarrow BR^i(\tau, U) \vee K^{i+1}$ 
  until check( $\neg(BR^{i+1}(\tau, U) \rightarrow BR^i(\tau, U))$ ) = unsat
  return safe
end

```

Figure 1: Backward Reachability for Array-based Systems.

the array sort to model the processes locations and more generally local data (like clocks, tickets, etc). Local data and processes locations can in fact be described as functions (aka arrays) mapping indexes to suitable elements.

More generally, an array-based system is specified once a tuple \mathbf{v} of variables (of index, array or elements sorts) is given, together with a formula $I(\mathbf{v})$ specifying initialization and with a finite sets of formulae $\tau_h(\mathbf{v}, \mathbf{v}')$ specifying system evolution (here the \mathbf{v}' are renamed copies of the \mathbf{v} , as usual in model-checking notation). Below, we let $\tau := \bigvee \tau_h$ and $\text{Pre}(\tau, K(\mathbf{v})) := \exists \mathbf{v}' (\tau(\mathbf{v}, \mathbf{v}') \wedge K(\mathbf{v}'))$; the formula $\text{Pre}(\tau, K(\mathbf{v}))$ describes the preimage of K , i.e. the set of states that can reach in one step a state satisfying K .

Bad configurations are described via a formula $U(\mathbf{v})$. Thus a safety problem becomes a tuple

$$\mathcal{S} = \langle \mathbf{v}, I(\mathbf{v}), \{\tau_h(\mathbf{v}, \mathbf{v}')\}_{h=1, \dots, t}, U(\mathbf{v}) \rangle \quad (1)$$

and backward reachability search can be implemented by the algorithm in Figure 1. The algorithm repeatedly computes pre-images of the set of bad states, until either a fixed point is reached (the system is ‘safe’) or until the intersection with initial states is not empty (the system is ‘unsafe’ in this case, i.e. bad configurations can be reached).

Clearly, for the algorithm to be effective, one should be able to discharge the required consistency tests; these tests involve a suitable version T_{arr} of the theory of arrays (the precise identification of this theory depends on the problem and should be specified together with \mathcal{S}). We have *safety tests* at lines 3, 6 and *fixpoint tests* at line 9; their effectiveness depends on T_{arr} and on the shape of the formulae I, τ_h, U .

In parameterized distributed systems, there is no arithmetic on index sort (processes are just ordered) and T_{arr} is quite simple (see [35]); in this situation, if we view the system variables \mathbf{v} as fresh constants, configurations can be identified with finitely generated models of T_{arr} (with generators having all index sort), ordering among configurations is model-theoretic embeddability and upward sets are characterizable via definability with existential sentences (i.e. sentences obtained by prefixing a string of existential index quantifiers to quantifiers-free formulae). We thus obtain a reformulation of the framework of well-structured systems [2] in a purely logical context, using basic model-theoretic ingredients.

Now, if U is existential and if the preimage of an existential sentence is existential, we always generate existential sentences in backward search. Since usually the initial formula I is universal (i.e. its negation is existential), fixpoint and safety tests only require testing consistency of the conjunction of an

existential sentence with a universal one; this is decidable if T_{arr} is sufficiently simple. So *the only problem is to guarantee that the preimage of an existential formula along a transition τ_h is still existential*. Here we re-encounter (mutatis mutandis) the problem of universal conditions in guards.

The problem does not arise if the transition formulae are guarded assignments in purely functional forms

$$\exists \underline{i} (\phi(\underline{i}, \mathbf{v}) \wedge \mathbf{v}' = F(\underline{i}, \mathbf{v})) \quad (2)$$

Here \underline{i} is a tuple of existentially quantified index variables, ϕ is quantifier-free and F is a quantifier-free definable function (we do not enter into technical details, but we point out that with such F you can express array updates making use of case-distinctions). Now, transitions like (2) can easily express e.g. broadcast and synchronizations actions, but they are insufficient to formalize for instance a basic ‘bakery’ mutual exclusion protocol. To formalize the latter, one needs universal quantifiers in guards, i.e. we need the following format

$$\exists \underline{i} (\phi(\underline{i}, \mathbf{v}) \wedge \forall k \psi(k, \underline{i}, \mathbf{v}) \wedge \mathbf{v}' = F(\underline{i}, \mathbf{v})) \quad (3)$$

as shown in the example below.

Example. We consider a protocol (taken from [1]) ensuring mutual exclusion for an arbitrary number of processes with a linear topology. Each process has four control locations: I(dle), R(equesting), W(aiting), and C(ritical section). The processes are linearly ordered: the set of processes on the left (right) of a given process p are those processes coming before (after, respectively) p in the linear order. Initially, all processes are in control location I. Each process can perform the following transitions:

- t_1 : if a process is in location I and all other processes (on its left and right) are either in location I or R, then it can move to location R;
- t_2 : if a process is in location R, then it can move to location W;
- t_3 : if a process is in location W and all the processes on its left are in location I, then it can move to location C;
- t_4 : if a process is in location C, then it can move to location R; and
- t_5 : if a process is in location R, then it can move to location I.

All those (bad) states containing at least two distinct processes in the control location C violate the mutual exclusion property that the protocol is intended to guarantee.

To formalize this protocol as an array-based system, we use as \mathbf{v} the single array variable a ; the theory T_{arr} is a very weak fragment of standard array theories; we specify I, τ_h, U below.

The set of initial states is characterized by

$$I(a) := \forall z. a[z] = I,$$

and the transitions are formalized by

$$\begin{aligned} \tau_1(a, a') &:= \exists i. (a[i] = I \wedge \forall j. (j \neq i \Rightarrow a[j] \in \{I, R\})) \wedge a' = \text{upd}(a, i, R) \\ \tau_2(a, a') &:= \exists i. (a[i] = R \wedge a' = \text{upd}(a, i, W)) \\ \tau_3(a, a') &:= \exists i. (a[i] = W \wedge \forall j. (j < i \Rightarrow a[j] = I) \wedge a' = \text{upd}(a, i, C)) \\ \tau_4(a, a') &:= \exists i. (a[i] = C \wedge a' = \text{upd}(a, i, R)) \\ \tau_5(a, a') &:= \exists i. (a[i] = R \wedge a' = \text{upd}(a, i, I)), \end{aligned}$$

where $a[j] \in \{I, R\}$ abbreviates $a[j] = I \vee a[j] = R$ and $\text{upd}(a, i, c)$ abbreviates the case-defined function $\lambda j. (\text{if } (j = i) \text{ then } c \text{ else } a[j])$ for a, i , and c constants of appropriate sorts. Notice that τ_2, τ_4, τ_5 have the form (2) but τ_1, τ_3 have the form (3).

The formula describing the set of bad states is

$$U(a) := \exists i, j. (i < j \wedge a[i] = C \wedge a[j] = C).$$

Notice that U is existential. ⊣

Since the preimage along transitions having the form (3) do not yield existential formulae, we modify them by monotonic abstraction. The idea is to re-interpret monotonic abstraction in the following way (which seems to be loosely motivated by the ‘stopping failures’ paradigm in distributed algorithm literature [37]). We assume that processes can crash at any instant of time and that crashed processes do not take part anymore to the protocol (in the terminology of Section 2 we are removing them, by passing to a ‘sub-configuration’). In this setting, a transition of the form (3) can always fire, provided the processes violating the universal guard $\forall k \psi(k, \underline{i}, \mathbf{v})$ crash. This transformation can be interpreted as a modification of the underlying computational model (we are adopting the ‘stopping failures’ paradigm) or more simply just as a kind of abstraction. One should be aware that the modified system has more runs, so safety of the modified system implies safety of the original one but not vice versa. Whatever it is, the point is that in the context of array-based system, *this monotonic abstraction modification can be performed at the syntactic level*: by using quantifier relativizations and by adding a ‘crash’ case to the update function F , it is possible to transform a transition τ_h having the form (3) into a transition $\hat{\tau}_h$ having the form (2) (details are fully explained in [17]).

From the experimental viewpoint, the above approach was successfully implemented in the tools MCMT and SAFARI [14]. Since its first releases, MCMT showed very good results in the verification of several classes of benchmarks taken from heterogeneous application domains (timed automata, broadcast protocols, cache coherence protocols). Two significant case studies [16, 26] provided empirical evidence of the effectiveness of this approach on non-trivial problems. We acknowledge also a more recent re-implementation of MCMT on a parallel architecture in the tool CUBICLE [28]. With the help of further advanced heuristics, CUBICLE outperformed previous tools and obtained impressive results [29] (including the first completely automatic parameterized verification of the ‘Flash’ cache coherence protocol). Broadening the horizon of possible applications, the MCMT approach has been adopted in [11, 12] to check the absence of flaws in Role-Based Access Control policies.

4 Array Acceleration

Let us examine more closely the effect of the syntactic monotonic abstraction of the previous section. If we take an existential formula K and a transition τ_h of the form (3), the preimage $\text{Pre}(\tau_h, K)$ has the form

$$\exists \underline{i} \forall k \psi(\mathbf{v}, \underline{i}, k), \quad (4)$$

where ψ is quantifier-free. This formula can be overapproximated by an existential formula by taking

$$\exists \underline{i} \bigwedge_t \psi(\mathbf{v}, \underline{i}, t), \quad (5)$$

varying t among a set of terms X . We may call (5) a *monotonic abstraction of the formula* (4) (notice that this notion is relative to X). There is an interesting relationship between the syntactic monotonic abstraction for transitions introduced in the previous section and the above monotonic abstraction for formulae. Indeed, if one take the obvious choice $X := \underline{i}$, it turns out that $\text{Pre}(\hat{\tau}_h, K)$ is precisely (5). Thus, monotonic abstraction can be applied at run-time (i.e. during backward search) and not in a preprocessing step (replacing the τ_h by the corresponding $\hat{\tau}_h$). This observation makes the technique of monotonic abstraction very flexible: one may choose X depending on the formula (4), according to some heuristics.

Let us now explain how monotonic abstraction can be imported in software model checking. First, notice that we can easily translate a sequential program manipulating integers and array variables into an array-based system. We get some simplifications with respect to distributed systems: first, initial and unsafe formulae are quite simple, they just say that a specific integer variable $pc \in \mathbf{v}$ (the ‘program counter’) is equal to the initial and to the error locations, respectively. There are no quantifiers involved here and there are no quantifiers in transitions too. In fact, transitions are ground assignments of the form

$$\phi(\mathbf{v}) \wedge \mathbf{v}' = F(\mathbf{v}) \quad (6)$$

corresponding to the various instructions of an imperative program. Fixpoint and safety tests consequently involve just ground formulae, so they are easily seen to be decidable.

However, the array theory T_{arr} needed is now more complicated, because we have some arithmetic on indexes (usually we need Presburger arithmetic, but often difference logic is sufficient). The complications in the array theory compromise termination analysis, because it is not possible anymore to get a WQO among configurations (whatever ‘configuration’ may mean here). In fact, termination is the major source of problems, even from the practical point of view. Let us show what happens in an example.

Example. The following ‘initialize-and-test’ simple example is quoted as problematic for CEGAR techniques in [36]:

```
for(I=0; I!= a_length; I++) a[I]=0;
for(J=0; J!= a_length; J++) assert(a[J]==0);
```

We first translate the above pseudo-code into the formalism of array-based systems. We need two integer variables I, J , a program counter p and an array variable a . We have five transitions:

$$\begin{aligned} \tau_0 &= \left(\begin{array}{l} p = 0 \wedge p' = 1 \wedge \\ I' = 0 \wedge J' = J \wedge a' = a; \end{array} \right) \\ \tau_1 &= \left(\begin{array}{l} p = 1 \wedge I \neq a_length \wedge p' = 1 \wedge \\ I' = I + 1 \wedge J' = J \wedge a' = wr(a, I, 0); \end{array} \right) \\ \tau_2 &= \left(\begin{array}{l} p = 1 \wedge I = a_length \wedge p' = 2 \wedge \\ I' = I \wedge J' = 0 \wedge a' = a; \end{array} \right) \\ \tau_3 &= \left(\begin{array}{l} p = 2 \wedge J \neq a_length \wedge a[J] = 0 \\ p' = 2 \wedge I' = I \wedge J' = J + 1 \wedge a' = a; \end{array} \right) \\ \tau_4 &= \left(\begin{array}{l} p = 2 \wedge J \neq a_length \wedge a[J] \neq 0 \\ p' = 4 \wedge I' = I \wedge J' = J \wedge a' = a; \end{array} \right) \\ \tau_5 &= \left(\begin{array}{l} p = 2 \wedge J = a_length \wedge p' = 3 \wedge \\ I' = I \wedge J' = J \wedge a' = a; \end{array} \right) \end{aligned}$$

where $wr(a, i, e)$ represents a copy of the array a with the element e written in position i .

The system is initialized by $p = 0$ and the unsafe condition is unreachability of location 4, namely it is represented by the formula $p = 4$.

Backward search trivially diverges here, because it produces formulae like

$$\begin{aligned}
 & p = 2 \wedge J \neq a_length \wedge a[J] \neq 0 \\
 & p = 2 \wedge J + 1 \neq a_length \wedge a[J + 1] \neq 0 \wedge a[J] = 0 \\
 & \dots \\
 & p = 2 \wedge J + n \neq a_length \wedge a[J + n] \neq 0 \wedge \bigwedge_{k=J}^{J+n-1} a[k] = 0 \\
 & \dots
 \end{aligned}$$

⊥

To stop divergence, we need to re-introduce quantifiers. One possible solution is to summarize the effect of n executions of a loop into a single transition, representing transitive closure. This technique is known as *acceleration* in model-checking and has been extensively investigated for fragments of Presburger arithmetic: integer relations having definable transitive closure include relations that can be formalized as difference bounds constraints [23, 27], octagons [21] and finite monoid affine transformations [33] (the paper [22] presents a general approach covering all these domains).

In our setting, we need acceleration inside array theories: the subject is investigated in [18], where a class of “acceleratable” ground guarded assignments is identified. The crucial observation is that *accelerated transitions have the form (3)*, hence syntactic monotonic abstraction can be applied here. In the example above, we can accelerate transitions 1 and 3, resulting in

$$\begin{aligned}
 \tau_1^+ &= \exists n > 0 \left(p = 1 \wedge \forall k (I \leq k < I + n \rightarrow k \neq a_length) \wedge p' = 1 \wedge \right. \\
 &\quad \left. I' = I + n \wedge J' = J \wedge a' = wr(a, [I, I + n - 1], 0) \right); \\
 \tau_3^+ &= \exists n > 0 \left(p = 2 \wedge \forall k (J \leq k < J + n \rightarrow k \neq a_length \wedge a[k] = 0) \right. \\
 &\quad \left. \wedge p' = 2 \wedge I' = I \wedge J' = J + n \wedge a' = a \right).
 \end{aligned}$$

Applying acceleration and syntactic monotonic abstraction, non-termination can often be avoided: for instance, the ‘initialize-and-test’ problem above is solved easily and correct quantified invariants are synthesized.

Some remarks are in order, however, to point out some differences with respect to the framework of Section 3. First of all, the array theory T_{arr} used in this section is more powerful and as a consequence satisfiability tests are effective only for more restricted classes of formulae. A decidable class include existential formulae and this class is sufficient to discharge safety tests (because the initial formula is ground now - it just says that the program counter is initialized to the first line code location). For fixpoint tests, the situation is different because here we must test for satisfiability the conjunction of an existential formula with a universal one. The idea is to use some instantiation-based incomplete calculus; the effect is not tremendous, i.e., it does not compromise the soundness of the tool, because if the model-checker fails to recognize a fixpoint, it is subject to extra work and possibly to divergence, but it does not produce wrong safe/unsafe outputs for this reason.

There is also a good new, however. Monotonic abstraction is just a heuristics for abstraction here, it may not produce spurious answers, if handled carefully. In fact, the addition of accelerated transitions does not modify the semantics of the system (contrary to what happened with the adoption of the stopping failure model), so error traces containing accelerated transitions can simply be ignored (if the system is unsafe, there should be a way to discover it without using accelerated transitions). So the best way to exploit monotonic abstraction for accelerated transitions is to use this technique inside abstract/refinement cycles, as an additional machinery to produce abstractions.

Problem	kind	d	#n	#del	#SMT	#inv	#ref	heur	time
Illinois	(C)	4	8	0	212	0	0	-	0.06
German	(C)	26	2121	255	117121	0	0	-	60.76
German_buggy	(C)	16	1300	203	24275	0	0	-	14.28
Bakery	(M)	2	1	0	29	0	0	-	0.00
Szymanski	(M)	11	17	5	1092	12	0	I	0.21
Szymanski_atomic	(M)	19	63	7	5470	32	0	I	1.82
Distributed Lamport	(M)	23	248	42	19622	7	0	I	27.18
Crash	(D)	13	113	21	1731	0	0	-	0.75
Fischer	(D)	10	16	2	363	0	0	-	0.08
Fischer_buggy	(D)	6	16	0	307	0	0	-	0.06
Lynch-Shavit_full	(D)	25	1103	99	56638	0	0	-	33.39
Strcpy	(S)	4	4	2	48	0	0	A	0.01
Strcmp	(S)	6	10	4	128	0	0	A	0.02
Max_in_array	(S)	7	13	6	166	0	0	A	0.04
Reverse	(S)	4	8	5	101	0	0	A	0.03
Palindrome	(S)	4	7	4	107	0	0	A	0.04
AllDifferent	(S+)	7	49	39	871	0	8	A+AR	0.40
BubbleSort	(S+)	5	14	10	200	0	0	A	0.07
InsertionSort	(S+)	18	98	56	3874	0	2	AR	1.43
SelectionSort	(S+)	8	101	77	6059	8	11	AR+I	4.98

Table 1: MCMT Statistics. The experiments were run on a laptop Intel(R) Core(TM) i3 CPU 2.27GHz with 4GB RAM running Linux Ubuntu 12.04. In the second column we indicate the class of the problem: (M) mutual exclusion, (C) cache coherence, (D) other distributed protocols (timed, fault tolerant, etc.), (S) sequential program for arrays, (S+) sequential program for arrays with nested loops. In column 3-8 we respectively give the depth of the search tree, the number of nodes generated by the tool in the search tree, the number of subsumed or subcovered nodes, the number of calls to the SMT solver, the number of invariants found by the tool in forward search and the number of refinements applied in abstraction/refinement mode. In the last column we put the total time in seconds and in the last-but-one column a summary of the options used (A=acceleration, AR=abstraction/refinement, I=invariant search). For each problem we reported the result in the best configuration we found for the tool.

5 Implementations

A framework for abstraction/refinement based on interpolation was introduced in [13, 15] as an extension to array programs of McMillan technique [38]; the framework was first implemented in the SAFARI tool [14]. The model-checker MCMT, since version 2.0, implements both abstraction/refinement for arrays and array acceleration with monotonic abstraction. Merging the two techniques has considerably raised the number of benchmarks solved by the tool.

The MCMT architecture is rather complex, because it contains not only the interface with the underlying SMT-solver (which is YICES), but also various modules for acceleration, abstraction, quantifier instantiation and quantifier elimination. We refer the reader to [20], for a presentation of the recent version 2.5. Table 1 below summarizes some performances on well-known benchmarks. We underline that the verification problems are all solved in their *parametric* version, i.e. regardless to the dimension of

the system/length of arrays or strings.

It should be noticed that often array acceleration produces quantified formulae that falls within decidable classes like those studied in [19, 25]; thus, if the control flow of the program is flat, it is not convenient to apply monotonic abstraction. This is what happens inside BOOSTER¹. BOOSTER is an integrated framework providing, among other static analysis techniques, an implementation of the acceleration paradigm presented in [19], where acceleration is exploited *precisely* without the adoption of the monotonic abstraction technique. BOOSTER implements also heuristics to execute MCMT in a parallel way, in order to try different promising configurations of the tool.

Acknowledgements. The work of the first author was supported by the Swiss National Science Foundation under grant no. P1TIP2 152261 and the one of the second by Italian Ministry of Education, University and Research (MIUR) under the PRIN 2010-2011 project “Logical Methods for Information Management”.

References

- [1] P. A. Abdulla (2010): *Forcing Monotonicity in Parameterized Verification: From Multisets to Words*. In: *Proceedings of SOFSEM '10*, Springer-Verlag, pp. 1–15, doi:10.1007/978-3-642-11266-9_1.
- [2] P. A. Abdulla, K. Cerans, B. Jonsson & Y.-K. Tsay (1996): *General Decidability Theorems for Infinite-State Systems*. In: *Proc. of LICS*, pp. 313–321, doi:10.1109/LICS.1996.561359.
- [3] P. A. Abdulla, N. B. Henda, G. Delzanno & A. Rezzine (2008): *Handling Parameterized Systems with Non-atomic Global Conditions*. In: *Proc. of VMCAI, LNCS 4905*, pp. 22–36, doi:10.1007/978-3-540-78163-9_7.
- [4] P. A. Abdulla & B. Jonsson (1993): *Verifying Programs with Unreliable Channels*. In: *LICS*, pp. 160–170, doi:10.1109/LICS.1993.287591.
- [5] P. A. Abdulla & A. Nylén (2001): *Timed Petri nets and BQOs*. In: *ICATPN*, pp. 53–70, doi:10.1007/3-540-45740-2_5.
- [6] P.A. Abdulla, M. Atto, J. Cederberg & R. Ji (2009): *Automatic Verification of Dynamic Data-Dependent Programs*. In: *ATVA, LNCS*, doi:10.1007/978-3-642-04761-9_16.
- [7] P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza & A. Rezzine (2008): *Monotonic Abstraction for Programs with Dynamic Memory Heaps*. In: *CAV*, pp. 341–354, doi:10.1007/978-3-540-70545-1_33.
- [8] P.A. Abdulla, G. Delzanno, N.B. Henda & A. Rezzine (2007): *Regular Model Checking Without Transducers*. In: *TACAS, LNCS 4424*, pp. 721–736, doi:10.1007/978-3-540-71209-1_56.
- [9] P.A. Abdulla, G. Delzanno & A. Rezzine (2007): *Parameterized Verification of Infinite-State Processes with Global Conditions*. In: *CAV, LNCS*, pp. 145–157, doi:10.1007/978-3-540-73368-3_17.
- [10] P.A. Abdulla, G. Delzanno & A. Rezzine (2008): *Monotonic Abstraction in Parameterized Verification*. In: *RP, ENTCS*.
- [11] F. Alberti, A. Armando & S. Ranise (2011): *ASASP: Automated Symbolic Analysis of Security Policies*. In: *CADE*, pp. 26–33, doi:10.1007/978-3-642-22438-6_4.
- [12] F. Alberti, A. Armando & S. Ranise (2011): *Efficient symbolic automated analysis of administrative attribute-based RBAC-policies*. In: *ASIACCS*, pp. 165–175, doi:10.1145/1966913.1966935.
- [13] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise & N. Sharygina (2012): *Lazy Abstraction with Interpolants for Arrays*. In: *LPAR*, pp. 46–61, doi:10.1007/978-3-642-28717-6_7.
- [14] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise & N. Sharygina (2012): *SAFARI: SMT-Based Abstraction for Arrays with Interpolants*. In: *CAV*, doi:10.1007/978-3-642-31424-7_49.

¹Available at www.inf.usi.ch/phd/alberti/prj/booster.

- [15] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise & N. Sharygina (2014): *An extension of Lazy Abstraction with Interpolation for programs with arrays*. *Formal Methods in System Design* 45(1), pp. 63–109, doi:10.1007/s10703-014-0209-9.
- [16] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise & G. P. Rossi (2010): *Brief Announcement: Automated Support for the Design and Validation of Fault Tolerant Parameterized Systems - A Case Study*. In: *DISC*, pp. 392–394, doi:10.1007/978-3-642-15763-9_36.
- [17] F. Alberti, S. Ghilardi, E. Pagani, S. Ranise & G.P. Rossi (2012): *Universal Guards, Relativization of Quantifiers, and Failure Models in Model Checking Modulo Theories*. *JSAT* 8(1/2), pp. 29–61.
- [18] F. Alberti, S. Ghilardi & N. Sharygina (2013): *Definability of Accelerated Relations in a Theory of Arrays and Its Applications*. In: *FroCoS*, pp. 23–39, doi:10.1007/978-3-642-40885-4_3.
- [19] F. Alberti, S. Ghilardi & N. Sharygina (2014): *Decision Procedures for Flat Array Properties*. In: *TACAS*, pp. 15–30, doi:10.1007/978-3-642-54862-8_2.
- [20] F. Alberti, S. Ghilardi & N. Sharygina (2014): *A framework for the verification of parameterized infinite-state systems*. In: *CILC*, CEUR.
- [21] M. Bozga, C. Girlea & R. Iosif (2009): *Iterating octagons*. In: *TACAS*, LNCS, pp. 337–351, doi:10.1007/978-3-642-00768-2_29.
- [22] M. Bozga, R. Iosif & F. Konecny (2010): *Fast Acceleration of Ultimately Periodic Relations*. In: *CAV*, LNCS, doi:10.1007/978-3-642-14295-6_23.
- [23] M. Bozga, R. Iosif & Y. Lakhnech (2009): *Flat parametric counter automata*. *Fundamenta Informaticae* (91), pp. 275–303, doi:10.3233/FI-2009-0044.
- [24] A.R. Bradley & Z. Manna (2007): *Calculus of computation: decision procedures with applications to verification*. Springer, doi:10.1007/978-3-540-74113-8.
- [25] A.R. Bradley, Z. Manna & H.B. Sipma (2006): *What’s Decidable About Arrays?* In: *VMCAI*, pp. 427–442, doi:10.1007/11609773_28.
- [26] R. Bruttomesso, A. Carioni, S. Ghilardi & S. Ranise (2012): *Automated Analysis of Parametric Timing-Based Mutual Exclusion Algorithms*. In: *NASA Formal Methods*, pp. 279–294, doi:10.1007/978-3-642-28891-3_28.
- [27] H. Comon & Y. Jurski (1998): *Multiple Counters Automata, Safety Analysis and Presburger Arithmetic*. In: *CAV*, LNCS 1427, Springer, pp. 268–279, doi:10.1007/BFb0028751.
- [28] S. Conchon, A. Goel, S. Krsti, A. Mebsout & F. Zadi (2012): *Cubicle: a Parallel SMT-based Model-Checker for Parameterized Systems*. In: *Proc. of CAV*, LNCS, doi:10.1007/978-3-642-31424-7_55.
- [29] S. Conchon, A. Goel, S. Krsti, A. Mebsout & F. Zadi (2013): *Invariants for Finite Instances and Beyond*. In: *Proc. of FMCAD*.
- [30] G. Delzanno (2000): *Automatic verification of parameterized cache coherence protocols*. In: *Proc. of CAV*, LNCS 1855, doi:10.1007/10722167_8.
- [31] E.A. Emerson & K.S. Namjoshi (1998): *On model checking for non-deterministic infinite-state systems*. In: *LICS*, p. 7080, doi:10.1109/LICS.1998.705644.
- [32] J. Esparza, A. Finkel & R. Mayr (1999): *On the Verification of Broadcast Protocols*. In: *Proc. of LICS*, IEEE Computer Society, pp. 352–359, doi:10.1109/LICS.1999.782630.
- [33] A. Finkel & J. Leroux (2002): *How to compose Presburger-accelerations: Applications to broadcast protocols*. In: *FST TCS 02*, Springer, pp. 145–156, doi:10.1007/3-540-36206-1_14.
- [34] S. Ghilardi, E. Nicolini, S. Ranise & D. Zucchelli (2008): *Towards SMT Model Checking of Array-Based Systems*. In: *IJCAR*, pp. 67–82, doi:10.1007/978-3-540-71070-7_6.
- [35] S. Ghilardi & S. Ranise (2010): *Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis*. *LMCS* 6(4), doi:10.2168/LMCS-6(4:10)2010.

- [36] R. Jhala & K.L. McMillan (2007): *Array Abstractions from Proofs*. In: CAV, pp. 193–206, doi:10.1007/978-3-540-73368-3_23.
- [37] Nancy A. Lynch (1996): *Distributed Algorithms*. Morgan Kaufmann.
- [38] K.L. McMillan (2006): *Lazy Abstraction with Interpolants*. In: CAV, pp. 123–136, doi:10.1007/11817963_14.
- [39] S. Toueg & T. D. Chandra (1990): *Time and Message Efficient Reliable Broadcast*. In: *Proc. 4th Int. Workshop on Distributed Algorithms*, LNCS, pp. 289–303, doi:10.1007/3-540-54099-7_20.